Procedia Computer Science

Volume 51, 2015, Pages 256–265

ICCS 2015 International Conference On Computational Science

Higher-Order Discrete Adjoint ODE Solver in C++ for Dynamic Optimization

Johannes Lotz[1], Uwe Naumann[1],
Ralf Hannemann-Tamás[2], Tobias Ploch[2], and Alexander Mitsos[2]

[1] RWTH Aachen, Software and Tools for Computational Engineering, Aachen, Germany
[lotz, naumann]@stce.rwth-aachen.de
[2] RWTH Aachen, Process Systems Engineering, Aachen, Germany
[Ralf.Hannemann-Tamas, Tobias.Ploch, Alexander.Mitsos]@avt.rwth-aachen.de

Abstract
Parametric ordinary differential equations (ODE) arise in many engineering applications. We consider ODE solutions to be embedded in an overall objective function which is to be minimized, e.g. for parameter estimation. For derivative-based optimization algorithms adjoint methods should be used. In this article, we present a discrete adjoint ODE integration framework written in C++ (`NIXE 2.0`) combined with Algorithmic Differentiation by overloading (`dco/c++`). All required derivatives, i.e. Jacobians for the integration as well as gradients and Hessians for the optimization, are generated automatically. With this framework, derivatives of arbitrary order can be implemented with minimal programming effort. The practicability of this approach is demonstrated in a dynamic parameter estimation case study for a batch fermentation process using sequential method of dynamic optimization. Ipopt is used as the optimizer which requires second derivatives.

Keywords: Adjoints, ODE, Algorithmic Differentiation, Dynamic Optimization, C++

1 Introduction

Technical or physical systems are often modeled by means of parametric ordinary differential equations (ODE). In this article, we consider parameter estimation with ODEs embedded. The structure of the objective function Φ is shown and explained in Figure 1. If the minimization is carried out by a second-order derivative-based method, the numerical optimizer requires first and second derivatives of the objective with respect to the free parameters.

According to the chain rule of differential calculus, derivatives of the objective naturally include derivatives of the embedded ODE solutions. Gradients and Hessians can be computed efficiently by first- and second-order adjoints, respectively [5, 17]. In this article we apply the discrete adjoint approach based on Algorithmic Differentiation (AD) [5, 13]. Our AD approach is modularized and flexible in that adjoint subroutines can be generated automatically

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.237

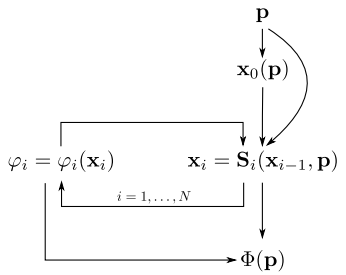

Figure 1: Illustration of the structure of the objective function Φ, which will also be used in the parameter estimation case study in Section 5. The parameters $\mathbf{p} \in \mathbb{R}^{n_p}$ enter together with state $\mathbf{x}_{i-1} \approx \mathbf{x}(t_{i-1}) \in \mathbb{R}^{n_x}$ the i-th embedded ODE solver call $S_i : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$, $i = 1, ..., N$. The solver is called N times in a loop, where for each intermediate state $\mathbf{x}_i$ a contribution $\varphi_i(\mathbf{x}_i)$ is computed. Those contributions are then used to form the objective Φ.

or optionally be coded manually. For this purpose, the C++ adjoint ODE solver `NIXE 2.0`[1] is combined with the AD tool `dco/c++`[2].

We consider parametric initial value problems (IVPs) of type

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \mathbf{p}) \ , \quad t \in [t_0, t_f] \ , \quad \mathbf{x}(t_0) = \mathbf{x}_0(\mathbf{p}) \ , \tag{1}$$

where $\mathbf{x}(t) \in \mathbb{R}^{n_x}$ is the state vector, $\dot{\mathbf{x}}(t)$ its time derivative, t is the independent time variable, $\mathbf{p} \in \mathbb{R}^{n_p}$ is a time-invariant parameter vector, t_0 and t_f are the initial and final times, respectively. The functions $\mathbf{f} : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$ and $\mathbf{x}_0 : \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$ provide the right hand side of the ODE and the initial values, respectively. They are assumed to be $\kappa + J$ and κ times continuously differentiable, respectively, with $\kappa \geq 2$. Here $J \geq 1$ denotes the order of the integration method for the numerical solution of the IVP. Under weak assumptions on $\mathbf{f}$ and $\mathbf{x}_0$, the solution $\mathbf{x}(t, \mathbf{p})$ is κ times continuously differentiable with respect to $\mathbf{p}$, which we assume in the following. For details on the existence theorems and mathematical properties of ODE systems we refer to [6]. In addition, we assume the overall objective function to be $\kappa \geq 2$ times continuously differentiable.

`NIXE` implements the solution of (1), but also a modified discrete adjoint method. The genericity of C++ facilitates the seamless coupling of AD libraries with `NIXE`. This combination enables arbitrary-order adjoint evaluation of the solver. Though the discrete approach is pursued in this article, it needs to be mentioned that there is ongoing research in discrete (e.g. [1]) and continuous (e.g. [10, 17]) approaches.

2 The Integrator `NIXE`

`NIXE` is a C++ template library for the solution of (1). It can be used both for the simulation and for adjoint sensitivity analysis based on modified discrete adjoints. The `NIXE` solver relies on the extrapolation of the linearly-implicit Euler discretization [4, 7] motivating the abbreviation `NIXE` for **N**IXE **I**s e**X**trapolated **E**uler. The concepts of the algorithm are briefly sketched in the following.

`NIXE` is an error-controlled variable step size and variable-order integrator. In general, we integrate from $t = t_0$ to $t = t_N = t_f$ with additional integrator stops due to the definition of the objective function on the grid points $t_1, t_2, \ldots, t_{N-1}$, see Figure 1. Since we use variable step size integration, within an interval $[t_{i-1}, t_i]$, there may be additional integration stops at $t_{i-1} = \tau_0^i, \tau_1^i, \ldots, \tau_{f_i-1}^i, \tau_{f_i}^i = t_i$, as illustrated in Figure 2. At each of these intermediate stops, we get accurate approximations of the state trajectories $\mathbf{x}(\tau_j^i)$ with an error within integration tolerance [6]. Let $\mathbf{y}_0$ be such an approximation. Then we compute the approximation $\mathbf{y}_1$ of

[1] `NIXE 2.0` is the successor of `NIXE 1.0`, which was first presented in [8].

[2] see `https://www.stce.rwth-aachen.de/software/dco_cpp.html`

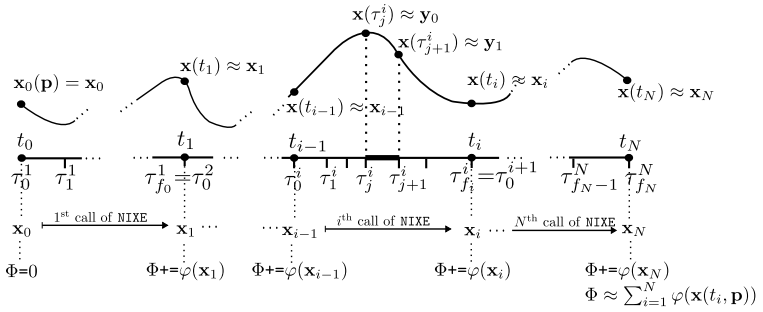

Figure 2: Time integration. The numerical integrator `NIXE` computes approximations $\mathbf{x}_i$ of the continuous solution trajectory $\mathbf{x}(t)$ at the measurement times t_i, $i = 1, \ldots, N$, with intermediate stops at the times τ_j^i. At the times t_i, the objective function value Φ is incremented by $\varphi(\mathbf{x}_i)$.

$\mathbf{x}(\tau_{j+1}^i)$ with respect to the step size $H = \tau_{j+1}^i - \tau_j^i$. We sketch the step from τ_j^i to τ_{j+1}^i, or in other words from $\mathbf{y}_0$ to $\mathbf{y}_1$, respectively.

Let J be the maximal extrapolation order in this step. For $j = 1, 2, \ldots, J$ and an ordered sequence of increasing integers $n_1 < n_2 < \cdots < n_J$, we set $h_j := H/n_j$, $\mathbf{y}_{j,0} := \mathbf{y}_0$, and apply the scheme

$$\mathbf{y}_{j,k+1} = \mathbf{y}_{j,k} + (\mathbf{I} - h\,\mathbf{A}_0)^{-1} h_j\,\mathbf{f}(\tau_j^i + (k+1)h_j, \mathbf{y}_{j,k}, \mathbf{p}), \quad k = 0, \ldots n_j - 1 \quad j = 1, \ldots J\ , \tag{2}$$

where

$$\mathbf{A}_0 = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\tau_j^i, \mathbf{y}_0, \mathbf{p}) \quad \text{and} \quad \mathbf{I} \in \mathbb{R}^{n_x \times n_x} \text{ is the identity matrix.} \tag{3}$$

This discretization scheme is of order one. It provides us with a family of first-order approximations $\mathbf{y}_{j,n_j}$ for $\mathbf{y}_1$. Higher-order methods [4] can be derived based on the Neville-Aitken scheme

$$\mathbf{Y}_{j,1} = \mathbf{y}_{j,n_j}, \quad \mathbf{Y}_{j,k+1} = \mathbf{Y}_{j,k} + \frac{\mathbf{Y}_{j,k} - \mathbf{Y}_{j-1,k}}{(n_j/n_{j-k}) - 1}, \quad k = 1, \ldots, j-1, \quad j = 1, \ldots, J\ . \tag{4}$$

For each step, the basic step size H and the extrapolation order J are determined online by a combined step size and order control. The reader is referred to [4] or [7, pp. 131–141] for a more detailed presentation.

Since the linearly-implicit Euler discretization is a so-called W-method [7, p. 114], the Jacobian A_0 could also be replaced by an approximation [16]. Schlegel et al. [19] exploit the W-method property to efficiently compute the sensitivities $\frac{\partial \mathbf{x}}{\partial \mathbf{p}}(t, \mathbf{p}) \in \mathbb{R}^{n_x \times n_p}$ by solving the first-order sensitivity equations

$$\frac{\partial \dot{\mathbf{x}}}{\partial \mathbf{p}} = \frac{\partial f}{\partial \mathbf{x}}(t, \mathbf{x}(t), \mathbf{p})\,\frac{\partial \mathbf{x}}{\partial \mathbf{p}} + \frac{\partial f}{\partial \mathbf{p}}(t, \mathbf{x}(t), \mathbf{p}), \quad t \in [t_0, t_f], \quad \frac{\partial \mathbf{x}}{\partial \mathbf{p}}(t_0, \mathbf{p}) = \frac{dx_0}{d\mathbf{p}}(\mathbf{p}) \tag{5}$$

by means of a tailored linearly-implicit Euler extrapolation. Loosely spoken, Schlegel et al. [19] construct a modified internal numerical differentiation scheme (cf. [3]), in that they drop second derivatives of $\mathbf{f}$.

For adjoint sensitivity analysis an analogous reduction is possible. Hannemann et al. [8] construct a modified discrete adjoint scheme of the linearly-implicit extrapolated Euler discretization, where also second-order terms are dropped. This adjoint scheme has exactly the same discretization error as the forward approach of Schlegel et al. [19].

As already mentioned, `NIXE` implements the modified discrete adjoint scheme of [8]. For computing adjoint sensitivities, the states at the time points τ_j^i are stored in checkpoints. If

the maximal number of checkpoints on an integration interval $[t_i, t_{i+1}]$ is fixed a priori, then `NIXE` uses `revolve` [21] for optimal online checkpointing. For the adjoint sweep, `NIXE` requires adjoints of $\mathbf{f}$ of the ODE in (1), i.e.

$$\mathbf{x}_{(1)} = \left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(t, \mathbf{x}, \mathbf{p})\right)^T \cdot \mathbf{y}_{(1)}, \quad \mathbf{p}_{(1)} = \left(\frac{\partial \mathbf{f}}{\partial \mathbf{p}}(t, \mathbf{x}, \mathbf{p})\right)^T \cdot \mathbf{y}_{(1)}, \quad \mathbf{y}_{(1)} \in \mathbb{R}^{n_x}. \tag{6}$$

In (6) we already use the convention that adjoint variables or functions are marked by parenthesized subscripts with the differentiation order. In principle, the user of `NIXE` has to implement the Jacobian A_0 in (3) and the adjoint projections in (6) of $\mathbf{f}$. Through the use of AD, those derivatives can be generated automatically as shown in Section 3.

3 Algorithmic Differentiation and `dco/c++`

Algorithmic Differentiation (AD) [5, 13] is a semantic program transformation technique that yields robust and efficient derivative code. We follow the notation from [13]. For a given implementation of a k-times continuously differentiable function $\Phi : \mathbb{R}^{n_p} \to \mathbb{R}, y = \Phi(\mathbf{p})$ for $\mathbf{p} \in \mathbb{R}^{n_p}$ and $y \in \mathbb{R}$, AD generates implementations of corresponding *tangent* and *adjoint* models automatically. The tangent model computes directional derivatives $y^{(1)} = \frac{\partial \Phi}{\partial \mathbf{p}} \cdot \mathbf{p}^{(1)}$, whereas the adjoint model computes (projected) gradients $\mathbf{p}_{(1)} = \frac{\partial \Phi}{\partial \mathbf{p}}^T \cdot y_{(1)}$. The vectors $\mathbf{p}^{(1)}, \mathbf{p}_{(1)} \in \mathbb{R}^{n_p}$ and the scalars $y^{(1)}, y_{(1)} \in \mathbb{R}$ are first-order tangents and adjoints of input and output variables, respectively, see [13] for more details. The computational cost of evaluating either of both models is a constant multiple of the cost of the original function evaluation. This observation becomes particularly advantageous when computing the gradient of Φ for $n_p \gg 1$. In this case, instead of n_p evaluations of the tangent model to compute the gradient element by element, only one evaluation of the adjoint model is required. Nonetheless, it needs to be mentioned that the adjoint mode of AD usually requires a lot more memory, which makes further techniques like checkpointing necessary. Higher derivatives can be obtained by recursive instantiations of the tangent and adjoint models.

AD can be implemented either via source code transformation or operator overloading techniques. We use our AD overloading tool `dco/c++`. It has been successfully applied to a number of problems in, for example, computational fluid dynamics [18, 22]. As an introductory example we consider a generic implementation of a scalar univariate function $y = f(x)$ in C++ as follows:

```
template <typename T> void f(const T& x, T &y);
```

For a simple evaluation the generic function is instantiated with data type `double`. A basic tangent model can be generated through instantiation with the `dco/c++` data type `dco::gt1s<double>::type`. This generic tangent 1st-order scalar type consists of `value` and `derivative` components of base type `double`. The base type is passed as a template argument and defines how the internal arithmetic is performed. The generic implementation of the function remains unchanged. Directional derivatives (tangents) are propagated according to the chain rule. The following driver computes $y^{(1)} = \frac{\partial f}{\partial x}(x) \cdot x^{(1)}$ at $x = 2$ with $x^{(1)}, y^{(1)} \in \mathbb{R}$ and $x^{(1)} = 1$.

```
#include "dco.hpp"
int main() {
  dco::gt1s<double>::type x, y;
  dco::value(x) = 2;
  dco::derivative(x) = 1;
  f(x, y);
  double dydx = dco::derivative(y);
```

```
  ...
}
```

On the other hand, adjoint inputs $x_{(1)}$ are computed as functions of adjoint outputs $y_{(1)}$. The basic adjoint model of `dco/c++` uses the generic adjoint 1st-order scalar type `dco::ga1s<double>::type`. The implied reversal of the data flow [15] requires the overloading tool to generate an internal representation of the function evaluation, which is usually referred to as the *tape*. An adjoint computation consists of a recording step (building the tape) and an interpretation step (propagate adjoints backwards through the tape). The following code computes $x_{(1)} = \frac{\partial f}{\partial x}(x)^T \cdot y_{(1)}$ at $x = 2$ with $x_{(1)}, y_{(1)} \in \mathbb{R}$ and $y_{(1)} = 1$. Registration of the active input x triggers the generation of the (in this case global) tape.

```
#include "dco.hpp"
int main() {
  dco::ga1s<double>::global_tape = dco::ga1s<double>::tape_t::create();
  dco::ga1s<double>::type x, y;
  dco::value(x) = 2;
  dco::ga1s<double>::global_tape->register_variable(x);
  f(x, y);
  dco::derivative(y) = 1;
  dco::ga1s<double>::global_tape->interpret_adjoint();
  double dydx = dco::derivative(x);
  ...
}
```

Derivatives of arbitrary oder can be computed by nesting the `dco/c++` data types. For example `dco::ga1s<dco::gt1s<double>::type>::type` facilitates the computation of second derivatives of Φ as follows:

$$\mathbf{p}^{(2)}_{(1)} = y_{(1)} \cdot \frac{\partial^2 \Phi}{\partial \mathbf{p}^2} \cdot \mathbf{p}^{(2)} + \frac{\partial \Phi}{\partial \mathbf{p}}^T \cdot y^{(2)}_{(1)}, \quad \text{with } \; y_{(1)}, y^{(2)}_{(1)} \in \mathbb{R} \text{ and } \mathbf{p}^{(2)}_{(1)}, \mathbf{p}^{(2)} \in \mathbb{R}^{n_p} \; ; \tag{7}$$

see also Section 5.

`dco/c++` offers an interface for integrating adjoint code provided by the user into the interpretation of the tape. This often yields more efficient and robust code; see, for example, [14]. The solution proposed in the next section combines a potentially user-defined adjoint version of NIXE with a `dco/c++` tape.

4 AD-enabled `NIXE`

The key ingredient of AD-enabled `NIXE` are C++ templates. Two important prerequisites are needed. First, all floating point operations within `NIXE` are performed with a generic data type `T`, which enables execution of `NIXE` with `double` as well as `dco/c++` data types. Second, the IVP definition is generic with respect to the floating point data type `T`. The function `EvalF` needs to be implemented by the user to compute the right hand side $\mathbf{f}$ of the ODE (1).

```
template <typename T>
struct model {
  static void EvalF(...){ ... }
  ...
};
```

This model definition is passed to `NIXE` as a class template (rather than an already instantiated template class) to enable its instantiation with arbitrary data types. The main routine in `NIXE` is declared as follows.

```
template <template <typename> class MODEL, typename T> void nixe(...);
```

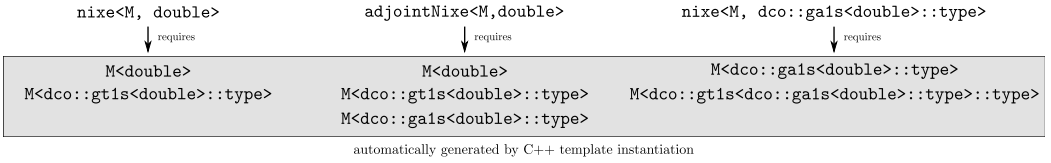

Figure 3: Instantiation graph for `nixe` and `adjointNixe`, the top-level subroutines of `NIXE` for simulation and modified discrete adjoint sensitivity analysis, respectively. `M` is the model class template described in Section 4. For a description of `dco/c++` data types see Section 3.

It is generic with respect to arithmetic type `T` and the generic IVP class definition. The arguments to the function include the parameters $\mathbf{p}$ and the previous state $\mathbf{x}_{i-1}$ as inputs and the current state $\mathbf{x}_i$ as output (see Figure 1).

Integration of the ODE in (1) requires the evaluation of $\mathbf{f}$ and of its Jacobian. Furthermore, the hand-written adjoint integrator `adjointNixe` (see Section 2) requires adjoints of $\mathbf{f}$. The instantiation graph with the corresponding `dco/c++` data types is shown in Figure 3. All required derivative models are generated automatically.

An adjoint of `NIXE` can now be computed either by calling the hand-written adjoint `adjointNixe` with the data type `double` or by calling the original function `nixe` with a `dco/c++` adjoint data type. Both versions should yield similar but not identical results, because `adjointNixe` is based on modified discrete adjoints [8] (see Section 2) that merely approximate the exact discrete adjoints. This difference can also be observed in the overall convergence behavior of the nonlinear solver for the parameter estimation problem in Section 5. For second-order optimization algorithms the Hessian of the objective including `NIXE` is required. It can be computed by instantiating the functions and classes in Figure 3 with a tangent data type instead of `double`. Nested templates in C++ allows us to add another order of derivative information to the model with minimal programming effort. Derivatives of arbitrary order can be implemented very elegantly. Both ways of adjoining `NIXE` are seamlessly integrated into an overall AD version of the objective function with `dco/c++`. Thus, AD can be applied to arbitrary objective functions with embedded calls of the ODE integrator.

5 Case Study

In the preceding sections we introduced a C++-framework for higher-order (modified) discrete adjoint sensitivity analysis for dynamic optimization. Dynamic parameter estimation of a batch fermentation process will serve as a case study to illustrate the application of the presented methodology. For nonlinear optimization, we use the software package `Ipopt`[3] [23]. An illustration of the calling structure is shown and described in Figure 4. The objective $\Phi : \mathbb{R}^{n_p} \to \mathbb{R}$ (see also Figure 1) collects discrepancies between simulated states $\mathbf{x}_i(\mathbf{p}) \approx \mathbf{x}(t_i, \mathbf{p}) \in \mathbb{R}^{n_x}$ and measurements $\hat{\mathbf{x}}_i \in \mathbb{R}^{n_x}$ of $\mathbf{x}(t)$ at t_i, where t_i, $i = 1, \ldots, N$, $t_1 < t_2 < \cdots < t_N = t_f$ correspond to measurement times:

$$\Phi(\mathbf{p}) := \sum_{i=1}^{N} \varphi_i(\mathbf{x}_i(\mathbf{p})) \ , \quad \varphi_i(\mathbf{x}_i(\mathbf{p})) := \|D \cdot (\mathbf{x}_i(\mathbf{p})) - \hat{\mathbf{x}}_i)\|_2^2 \, , \tag{8}$$

where $D = \mathrm{diag}(d_1, d_2, \ldots, d_{n_x}) \in \mathbb{R}^{n_x \times n_x}$ is a diagonal scaling matrix.

The considered reaction system describes the biotechnological production of itaconic acid

[3] see `https://projects.coin-or.org/Ipopt`

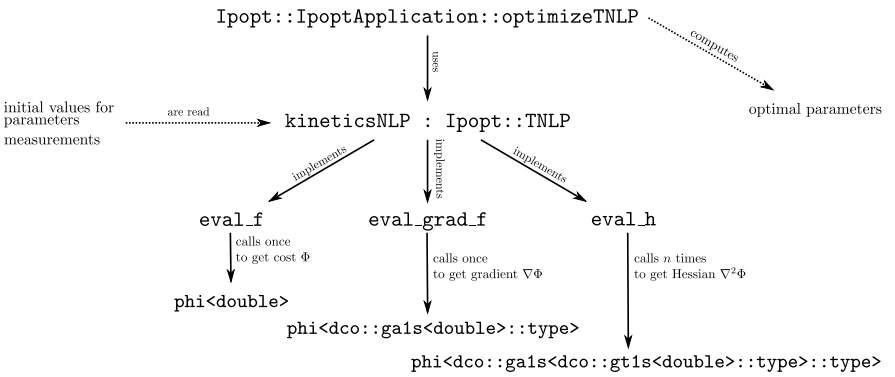

Figure 4: Parameter estimation framework with `Ipopt`. `template <typename T> phi` is a function template implementing the objective (8); different derivative information is obtained by instantiation with suitable `dco/c++` types.

using the fungus *Aspergillus terreus* (*A. terreus*) with glucose as substrate. This mechanism represents an important reaction step in the bio-based manufacturing of fine chemicals and fuels. An accurate and validated model is indispensable for further analysis and optimization of the fermentation process. The system can be described by modeling three components: mass of *A. terreus* (X), product (P), and substrate (S). For this case study, a Monod kinetic [12] was modified to model the specific growth rate of *A. terreus* similar to Song et al. [20]:

$$\mu_X = \mu_{\max} \frac{S}{S + K_S} \left(1 - \frac{P}{P_{\text{Crit}}}\right)^i , \tag{9a}$$

where μ_X is the specific growth rate, $\mu_{\max}$ is the maximum specific growth rate, S is the substrate concentration, K_S is the substrate saturation constant, P is the product concentration, and P_{Crit} is the critical product concentration. The exponent i describes the degree of product inhibition. With this growth rate, the kinetic equation for cell growth of *A. terreus* is

$$\dot{X} = \mu_X \, X \ . \tag{9b}$$

The formation of itaconic acid can be described by the Luedeking-Piret model [11]: $\dot{P} = \alpha \, \mu_X \, X + \beta \, X$. In this case, the formulation was modified to obtain a more accurate fit for product formation:

$$\dot{P} = \frac{1}{1 + \exp(\gamma \cdot (t_{\text{Lag}} - t))} \left(\alpha \, \mu_X \, X + \beta \, X\right) \left(1 - \frac{P}{P_{\text{Crit}}}\right)^i , \tag{9c}$$

where t_{Lag} describes the lag phase, γ is a smoothing coefficient, and α and β correspond to the Luedeking-Piret parameters. The product inhibition term is introduced to ensure that the product concentration does not exceed P_{Crit}. Substrate is converted to biomass (ratio Y_{XS}), product (ratio Y_{PS}) and also used as energy source for biomass (rate $m_X \, X$):

$$\dot{S} = -\frac{1}{Y_{\text{XS}}} \dot{X} - \frac{1}{Y_{\text{PS}}} \dot{P} - m_X \, X \ . \tag{9d}$$

The model is fitted to experimental data taken from Kuenz et al. [9]. The scaling values of $D = \text{diag}(d_X, d_P, d_S)$ in the objective function (8) are $d_X = \sqrt{10}$, $d_P = d_S = 1$. All fitting parameters and their final values are listed in Table 1. The measured values together with the initial and the fitted solutions are plotted in Figure 5.

parameter	unit	f. val.	i. gue.	parameter	unit	f. val.	i. gue.
$\mu_{\max}$	h^{-1}	0.288	0.061	K_S	$g \cdot L^{-1}$	1000	0.80
P_{Crit}	$g \cdot L^{-1}$	1000	87	i	-	85.6	1.4
γ	-	0.0952	0.2	t_{Lag}	h	93.3	10
α	$g \cdot g^{-1}$	5.45E-07	13.8	β	$g \cdot (g \cdot h)^{-1}$	24.4	0.035
Y_{XS}	$g \cdot g^{-1}$	0.818	0.1	Y_{PS}	$g \cdot g^{-1}$	0.709	2.0
m_X	$g \cdot (g \cdot h)^{-1}$	0.0150	0.0001	$P_{t=0}$	$g \cdot L^{-1}$	0	-*
$S_{t=0}$	$g \cdot L^{-1}$	180	-*	$X_{t=0}$	$g \cdot L^{-1}$	1	-*

Table 1: List of parameters used in the model equations with their corresponding units and fitted values (f. val.) and initial guesses (i. gue.). The lower and upper bounds of all optimization variables were set to $lb = 0$ and $ub = 1000$, respectively. $P_{t=0}$, $S_{t=0}$, and $X_{t=0}$ are the respective initial values for the components.
*Initial values are fixed and thus not optimized.

		run time	tape size	run time ratio
Φ		$7.1 \cdot 10^{-4} s$		
$\nabla\Phi$	`dco/c++` plain	$4.5 \cdot 10^{-3} s$	4500 kB	6.2
$\nabla^2\Phi \cdot \mathbf{v}$	`dco/c++` plain	$8.8 \cdot 10^{-3} s$	7000 kB	12
$\nabla\Phi$	`dco/c++` + hand-written	$4.8 \cdot 10^{-3} s$	2.7 kB	6.8
$\nabla^2\Phi \cdot \mathbf{v}$	`dco/c++` + hand-written	$9.8 \cdot 10^{-3} s$	4.3 kB	14

Table 2: Run time, tape size and run time ratio for single evaluation of Φ, $\nabla\Phi$ and $\nabla^2\Phi \cdot \mathbf{v}$ for different approaches of differentiation. The run time ratio is the run time divided by the run time of one evaluation of the objective without derivatives. The tape size measures only the memory footprint of `dco/c++`. Measurements are performed on an `Intel(R) Core(TM) i5-3337U`.

According to Section 4, gradients can be computed either by plain application of `dco/c++` to the complete implementation of the objective Φ including the function `nixe()`, or by calling the hand-written adjoint function `adjointNixe` described in Section 2, whenever adjoints of `NIXE` are required. Discrepancies in the derivatives result in inferior convergence for the manually written adjoint. With hand-written modified discrete adjoints, the optimizer Ipopt takes 74 major iterations until the stopping criterion is met, while with exact discrete adjoints, Ipopt requires only 63 major iterations. The slowdown of the convergence of nonlinear optimizers in presence of inexact derivatives is not surprising and has already been observed in comparison of AD with finite differences [2].

As shown in Table 2, hand-written modified and exact discrete adjoints of `NIXE` are comparable in terms of run time. The hand-written adjoint is expected to use much less memory in general, due to the approximations described in further detail in Section 2. The code for this case study including `NIXE` and a free trial license for `dco/c++` are available on request[4].

6 Conclusions

The results of the parameter fitting shown in Figure 5 look promising. However, the optimal parameter values deviate by orders of magnitude from those a system biologist might expect. Currently, model (9) should rather be identified as an empirical model than one that is based

[4]Please contact `info@stce.rwth-aachen.de`

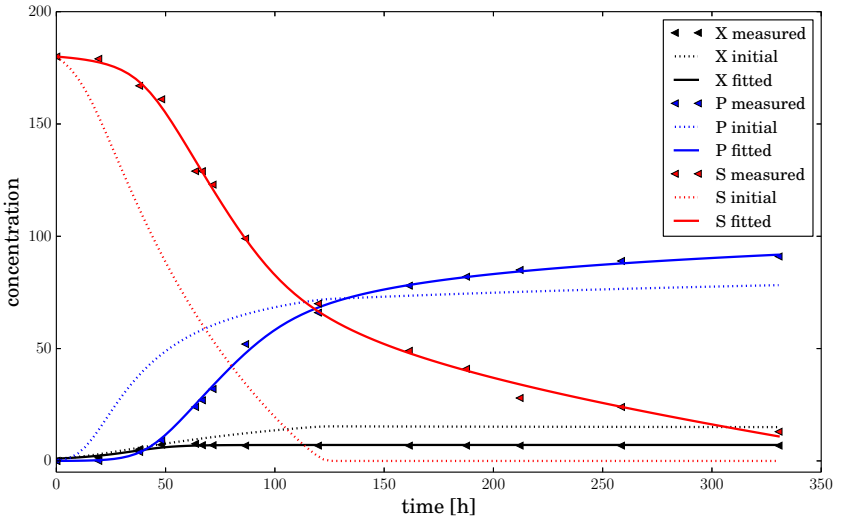

Figure 5: Measurements and simulated temporal evolution of the concentrations of the components mass of *A. terreus* (X), product (P), and substrate (S).

on first principle and constitutive equations. Thus, to find an accurate mechanistic model, there is still a lot of modeling and parameter estimation work to be done. To this end, it is very important to solve the parameter estimation problems very fast. We emphasize, that a large fraction of the overall computational effort is in the computation of the gradient and Hessian of the objective function. We were able to compute these derivatives very fast by coupling `NIXE` with the AD library `dco/c++`. The answer to the question, whether one should use exact or modified discrete adjoints of `NIXE` depends on some factors. Exact discrete adjoints have the advantage of providing exact derivative information of the discretized problem, which potentially leads to faster convergence of the optimizer. On the other hand the modified discrete adjoints have a significantly lower memory footprint.

Acknowledgments

We gratefully thank Jörn Viell for communicating the parameter estimation problem and Kirsten Ulonska for providing us with the initial model of the fermentation kinetics.

This work was financially supported by the BEProMod project, which is part of the NRW-Strategieprojekt BioSC funded by the Ministry of Innovation, Science and Research of the German State of North Rhine-Westphalia.

This work was performed as part of the Cluster of Excellence "*Tailor-Made Fuels from Biomass*", which is funded by the Excellence Initiative of the German federal and state governments to promote science and research at German universities.

References

[1] J. Albersmeyer and H. G. Bock. Sensitivity generation in an adaptive BDF-method. In H. G. Bock, E. Kostina, H. X. Phu, and R. Rannacher, editors, *Modeling, Simulation and Optimization*

of Complex Processes, pages 15–24. Springer Berlin Heidelberg, 2008.

[2] J.T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming*. Siam, 2001.

[3] H.G. Bock. *Numerical treatment of inverse problems in chemical reaction kinetics*. Springer, Heidelberg, 1981.

[4] P. Deufhard, E. Hairer, and J. Zugck. One-step and extrapolation methods for differential- algebraic systems. *Numer. Math.*, 51(5):501–516, 1987.

[5] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[6] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I - Nonstiff Problems*. Springer, Berlin, 1993.

[7] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II - Stiff and Differential-Algebraic Problems*. Springer, Berlin, 1996.

[8] R. Hannemann, W. Marquardt, B. Gendler, and U. Naumann. Discrete first- and second-order adjoints and automatic differentiation for the sensitivity analysis of dynamic models. In *Procedia Computer Science*, volume 1, pages 297–305, 2010.

[9] A. Kuenz, Y. Gallenmüller, T. Willke, and K.-D. Vorlop. Microbial production of itaconic acid: Developing a stable platform for high product concentrations. *Applied microbiology and biotechnology*, 96(5):1209–1216, 2012.

[10] S. Li and L. Petzold. Description of DASPKADJOINT: An adjoint sensitivity solver for differential-algebraic equations. Technical report, Department of Computer Science, University of California Santa Barbara, CA 93106, USA, Santa Barbara, CA, 2002.

[11] R. Luedeking and E. L Piret. A kinetic study of the lactic acid fermentation. Batch process at controlled pH. *Journal of Biochemical and Microbiological Technology and Engineering*, 1(4):393–412, 1959.

[12] J. Monod. The growth of bacterial cultures. *Annual Reviews in Microbiology*, 3(1):371–394, 1949.

[13] U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation*. Number 24 in Software, Environments, and Tools. SIAM, Philadelphia, PA, 2012.

[14] U. Naumann, J. Lotz, K. Leppkes, and M. Towara. Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations. *ACM Transactions on Mathematical Software*. To appear.

[15] U. Naumann, J. Utke, A. Lyons, and M. Fagan. Control Flow Reversal for Adjoint Code Generation. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 55–64. IEEE Computer Society, 2004.

[16] U. Nowak. Dynamic sparsing in stiff extrapolation methods. *IMPACT Comput. Sci. Eng.*, 5(1):53–74, 1993.

[17] D.B. Özyurt and P.I. Barton. Cheap second order directional derivatives of stiff ODE embedded functionals. *SIAM J. Sci. Comput.*, 26(5):1725–1743, 2005.

[18] M. Sagebaum, N. R. Gauger, U. Naumann, J. Lotz, and K. Leppkes. Algorithmic differentiation of a complex C++ code with underlying libraries. *Procedia Computer Science*, 18:208–217, 2013.

[19] M. Schlegel, W. Marquardt, R. Ehrig, and U. Nowak. Sensitivity analysis of linearly-implicit differential-algebraic systems by one-step extrapolation. *Appl. Numer. Math.*, 48(1):83–102, 2004.

[20] H. Song, S. H. Jang, J. M. Park, and S. Y. Lee. Modeling of batch fermentation kinetics for succinic acid production by *Mannheimia succiniciproducens*. *Biochemical Engineering Journal*, 40(1):107–115, 2008.

[21] P. Stumm and A. Walther. New algorithms for optimal online checkpointing. *SIAM J. Scientific Computing*, 32(2):836–854, 2010.

[22] M. Towara and U. Naumann. A discrete adjoint model for OpenFOAM. *Procedia Computer Science*, 18:429–438, 2013.

[23] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106:25–57, 2006.